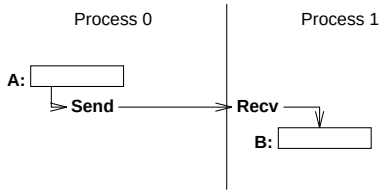

Figure 1: MPI messages.

1 MPI Hands-On - Sending and Receiving Messages I

Questions:

- To whom is data sent?
- What is sent?
- How does the receiver identify it?

1.1 Current Message-Passing

Message = data + envelope

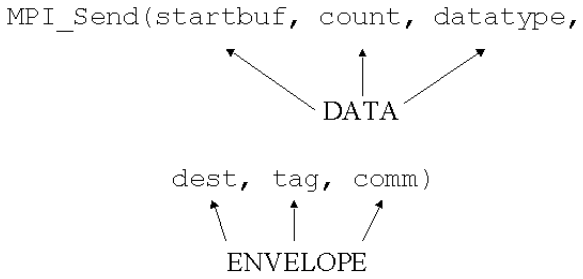

Figure 2: Data+Envelope.

- *MPI Data; Arguments*
  - **startbuf** (starting location of data)
  - **count** (number of elements)

    - receive count $\geq$ send count
  - **datatype** (basic or derived)
    - receiver datatype = send datatype (unless MPI_PACKED)
    - Elementary (all C types). Specifications of elementary datatypes allows heterogeneous communication.
    - MPI basic datatypes for C:

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Figure 3: MPI basic datatypes for C.

- *MPI Envelope; Arguments*
  - **destination or source**
    - rank in a communicator
    - receive = sender or MPI_ANY_SOURCE
  - **tag**
    - integer chosen by programmer
    - receive = sender or MPI_ANY_TAG (wild cards allowed)
  - **communicator**
    - defines communication "space"
    - group + context
    - receive = send
  - Collective operations typically operated on all processes.

- All communication (not just collective operations) takes place in groups.
- A context partitions the communication space. A message sent in one context cannot be received in another context. Contexts are managed by the system.
- A group and a context are combined in a communicator.
- Source/destination in send/receive operations refer to rank in group associated with a given communicator.

1.2 The Buffer

Sending and receiving only a contiguous array of bytes. Specified in MPI by *starting address*, *datatype*, and *count*

- hides the real data structure from hardware which might be able to handle it directly.
- requires pre-packing dispersed data
  - rows of a matrix stored columnwise.
  - general collections of structures.
- prevents communications between machines with different representations (even lengths) for same data type

1.3 MPI Basic Send/Receive

Thus the basic send (blocking!!) has become:

```
MPI_Send( start, count, datatype, dest, tag, comm )
```

and the receive (blocking!!):

```
MPI_Recv(start, count, datatype, source, tag, comm, status)
```

The source, tag, and count of the message actually received can be retrieved from `status`.

```
MPI_Status status;
MPI_Recv( ..., &status );
... status.MPI_TAG; ... status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &count );
```

MPI_Get_count may be used to determine how much data of a particular type was received.

Two simple collective operations (just to introduce!):

```
MPI_Bcast(start, count, datatype, root, comm)
MPI_Reduce(start, result, count, datatype,
           operation, root, comm)
```

1.4 Exercises/Examples

1. An example for communication world code1.c.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
  int size, my_rank;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  // printf("Executed by all processors: Hello! It is processor %d.\n
    ", my_rank);
  if (my_rank == 0)
  {
    printf("Hello! It is processor 0. There are %d processors in this
    comm. world.\n", size);
    printf ("I am process %i out of %i: Hello world!\n",my_rank, size)
    ;
  }
  else
  {
    printf("I am process %i out of %i: Hello world!\n", my_rank, size)
    ;
  }
  MPI_Finalize();
  return 0;
}
```

2. Write a program to send/receive and print out your name and age to each processors. code2.c.

```c
#include <stdio.h>
#include <mpi.h>
#include <string.h>

int main(int argc, char **argv)
{
  int my_rank; /* rank of process */
  int size; /* number of processes */
  int dest; /* rank of receiver */
  int my_age = 4; /* storage for my_age */
  char message[100]; /* storage for message */
  int recv_my_age = 0; /* storage for received my_age */
  MPI_Status status; /* return status for receive */

  MPI_Init(&argc, &argv); /* Start up MPI */
  MPI_Comm_size(MPI_COMM_WORLD,&size); /* Find out number of processes */
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank); /* Find out process rank */
  if(my_rank == 0)  /* rank of sender */
  {
    sprintf(message,"IKC-MH.57"); /* Create message */
    for(dest=1; dest<size; dest++)
    {
      printf("Sending to worker num:%d\n", dest);
      MPI_Send(&my_age, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
      MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, 2, MPI_COMM_WORLD);
    }
  }
  else
  {
    MPI_Recv(&recv_my_age, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(message, sizeof(message), MPI_CHAR, 0, 2, MPI_COMM_WORLD, &status);
    printf("==========\n");
    printf("I am node: %d\n", my_rank);
    printf("My age: %d\n", recv_my_age);
    printf("My name: %s\n", message);
    printf("==========\n");
  }
  MPI_Finalize(); /* Shut down MPI */
  return 0;
}
```